High-Performance Graphics 2023
J. Bikker and C. Gribble
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 8

Real-Time Ray Tracing of Micro-Poly Geometry with Hierarchical Level of Detail

Carsten Benthin Christoph Peters

Intel Corporation



Figure 1: Thai Statues *(200 M triangles),* Displaced Rungholt *(100 M triangles), and* Landscape *(132 M triangles) micro-poly geometries rendered with a diffuse path tracer at* 1920×1080 *with* > 20 *fps (1 spp, denoising included). All micro-poly geometries are clustered and compressed prior to rendering. For each frame a LOD subset of clusters (*$\sim 16-40$ *M triangles) are decompressed, converted and fused together into a BVH suitable for hardware-accelerated ray tracing. With a per-frame overhead of just* $5.7-9$ *ms, our approach is suitable for real-time applications.*

Abstract

In recent work, Nanite has demonstrated how to rasterize virtualized micro-poly geometry in real time, thus enabling immense geometric complexity. We present a system that employs similar methods for real-time ray tracing of micro-poly geometry. The geometry is preprocessed in almost the same fashion: Nearby triangles are clustered together and clusters get merged and simplified to obtain hierarchical level of detail (LOD). Then these clusters are compressed and stored in a GPU-friendly data structure. At run time, Nanite selects relevant clusters, decompresses them and immediately rasterizes them. Instead of rasterization, we decompress each selected cluster into a small bounding volume hierarchy (BVH) in the format expected by the ray tracing hardware. Then we build a complete BVH on top of the bounding volumes of these clusters and use it for ray tracing. Our BVH build reaches more than 74% of the attainable peak memory bandwidth and thus it can be done per frame. Since LOD selection happens per frame at the granularity of clusters, all triangles cover a small area in screen space.

1. Introduction

Real-time rendering has always been striving for greater geometric complexity to increase the fidelity of scenes. In recent years, real-time rasterization of micro-poly geometry with billions of virtualized triangles has become viable through *Nanite* [KSW21]. This system allows artists to feed extremely detailed geometry to the renderer. A preprocessing step partitions the geometry into clusters, generates a hierarchical level of detail (LOD) structure on top of these clusters and compresses them. At run time, the relevant LODs are streamed in, much like mipmaps for sparse virtual textures. Then clusters get selected for rasterization to a visibility buffer using occlusion culling and LOD heuristics.

Support for ray tracing in Nanite is rudimentary. It builds bottom-level acceleration structures (BLAS) on top of a low LOD of the Nanite meshes, independently of which LOD would be suitable for a particular view. In rasterization, Nanite typically renders around 20 million triangles for a single frame. Real-time ray tracing of a BLAS with that many triangles is viable. However, building the acceleration structure is a problem. APIs such as Vulkan [Khr20] and Direct3D 12 [Mic20] use top-level acceleration structures (TLAS) and BLAS, which can either be built from scratch or updated when only vertex positions have changed. Since the LOD mechanism in Nanite makes major changes to the mesh topology, the BLAS would need to be rebuilt completely each frame, which simply takes too much time. On our hardware, it would take 57-67 ms [BDTD22]. Building a BLAS for the highest LODs is not viable either because the memory footprint of a triangle in a BLAS is much bigger than that of a compressed triangle in a cluster. Most of the virtualized geometry will not fit into VRAM.

We propose a viable approach to render such micro-poly geometry with hierarchical LOD in real time using existing ray tracing hardware. Our method sidesteps the notion of monolithic BLAS builds by modifying the GPU-accelerated bounding volume hierarchy (BVH) builder in Embree 4 [Int23]. Because of that, our current implementation is specific to Intel hardware but with sufficiently low-level access to hardware BVH layouts, the same methods could work on any other GPU (Sec. 7). Much like Nanite, our system starts with extensive preprocessing of the input geometry (Sec. 3). First it partitions the geometry into clusters of up to 256 nearby triangles (Sec. 3.1). Then connected clusters are merged and their combined geometry is simplified without changing the boundary edges of the merged cluster (Sec. 3.2). Repeating this process yields hierarchical LOD, but eventually boundary edges become abundant and make simplification ineffective. In the spirit of Nanite, we occasionally split clusters to introduce new boundary edges, thus turning the LOD tree into an LOD directed acyclic graph (DAG) (Sec. 3.3). Finally, we use aggressive quantization to compress these clusters in a way that preserves water tightness (Sec. 3.4) and store them in a GPU-friendly data structure (Sec. 3.5).

The goal of the per-frame phase is to construct a BLAS with exactly the right LOD for a single frame (Sec. 4). First it selects the relevant clusters using heuristics for the appropriate LOD while obeying rules that guarantee a crack-free mesh (Sec. 4.1). Then a single pass over the selected clusters decompresses them into GPU's shared local memory, builds a small BVH per cluster and writes it to memory (Sec. 4.2). These small BVHs already use the format required by the ray-tracing hardware and are at their final memory locations. Finally, a BVH builder operates on top of the axis-aligned bounding boxes (AABB) of the clusters to build the upper levels of the BLAS (Sec. 4.3). At this point, the BLAS is ready to be used for ray tracing or path tracing.

Through this combination of preprocessing, compression and a BVH build on top of cluster AABBs, we reach unprecedented build speeds for BVHs (Sec. 5). We reach more than 74% of the memory bandwidth utilization of a thoroughly optimized MemCopy kernel. Thus, our work demonstrates that it is viable to generate a BLAS with exactly the right LOD for a single frame in real time. Of course our system is inferior to Nanite in many regards: The clustering, merging and splitting are less sophisticated and cannot reduce triangle counts as heavily. Geometry is not streamed in from disk, our LOD selection is less efficient and we do not have occlusion culling. Nonetheless, we demonstrate that hierarchical LOD in the spirit of Nanite is viable with existing ray tracing hardware.

2. Previous Work

Reducing memory consumption when ray tracing complex geometries has always been of high importance, and therefore been a research focus for decades. Various approaches either compress the geometric representation, introduce LOD or combine both.

Segovia et al. [SE10] propose an approach which reduces memory consumption of both BVH and geometry data. They apply hierarchical mesh quantization to combine BVH and triangle data into a single unified data structure. All vertices within a BVH leaf are quantized with respect to the leaf bounding box. Gaps due to different quantization reference points are avoided by snapping the vertices and leaf bounding boxes to a global grid over the scene.

Yoon et al. [YLM06] propose one of the first approaches that explores LOD generation for ray tracing. It uses a plane as distant LOD representation. In recent work, Ikeda et al. [IKH22] use AABBs of the BVH as proxy for distant LOD representation, as well as stochastic material sampling for material evaluation at any LOD level.

Another popular approach that explores LOD generation for ray tracing is the *Razor* system proposed by Djeu et al. [DHW*11]. The Razor system uses a multi-resolution geometry cache. Due to the high synchronization cost in coordinating shared cache accesses between different CPU threads, Djeu et al. propose that each CPU thread maintains its own cache independently from all other cores. However, a tessellation cache per thread suffers from redundant computation and data replication, thus leading to excessive memory requirements on architectures with large thread counts. Benthin et al. [BWN*15] introduce a method to cache tessellation data of subdivision surfaces without replication. The approach relies on adaptive subdivision of Catmull-Clark subdivision patches combined with lazy caching of tessellation results in a fixed-size cache, optimized for multi-core CPU rendering.

Tessellation data for subdivision surfaces or other structures with fine geometric detail can often be represented locally by a small grid-like topology (excluding adjacency information). Several works propose efficient methods for lossy compression of these grid-like topologies. For example, Benthin et al. [BVW21] encode the grid vertices as offset from a base primitive, i.e. a bilinear patch, using a reduced precision floating point representation.

NVIDIA's Ada Lovelace architecture introduces hardware support for ray tracing of displaced micro meshes [BM23]. A displaced micro mesh consists of base triangles with a displacement vector per vertex. Based on user-defined subdivision levels, the base triangles are hierarchically subdivided into micro triangles. The displacement vectors are interpolated across the base triangles, scaled by hierarchically encoded height values and applied as offsets to the micro vertices. This approach offers a compact geometry representation (1 to 8 bits per micro triangle) and an LOD mechanism. However, the used height fields within triangular prisms cannot capture arbitrary geometry since vector displacements are not supported. Additionally, the simplification cannot go lower than one micro triangle per base triangle. If a base mesh is inadequate or unavailable, it has to be generated from the high-resolution mesh.

In the realm of rasterization, Nanite [KSW21] makes real-time rendering of micro-poly geometry practical. This system starts with extensive per-mesh preprocessing: Initially, it creates clusters of up to 128 triangles, while optimizing for few boundary edges. Then it groups and merges clusters with many shared boundary edges together. Mesh simplification halves the triangle count of the merged clusters without changing their boundary edges. This process could be continued iteratively to get a hierarchical LOD tree. However, after a few steps, there will be too many boundary edges in comparison to interior edges (Figs. 3a, 3b). Nanite solves this problem through splitting: Simplified clusters are split into two clusters, each with at most 128 triangles, again while optimizing for few boundary edges. That introduces new, longer boundary edges and

turns the hierarchy into a DAG (Fig. 3c). The optimization problems for initial clustering, grouping and splitting are solved using graph partitioning but it is non-trivial to enforce the maximal triangle counts. Subsequently, clusters are compressed and stored in a format that enables streaming and GPU-driven rendering.

At run time, Nanite streams in the relevant pages of preprocessed geometry at the required LOD. It does occlusion culling using hierarchical depth buffers and visibility information from the previous frame. Clusters to display are selected in a parallel fashion using LOD heuristics. The heuristics produce a cut through the DAG, which guarantees complete and crack-free geometry (App. A). The selected clusters get rendered into a visibility buffer using a mixture of hardware and software rasterization. The visibility buffer is then turned into a G-buffer for deferred shading.

The preprocessing in our system has the same steps as Nanite but implements them differently. Since the AABBs of our clusters are eventually used in a BVH, clustering and grouping use heuristics of BVH builders. At run time, our system works quite differently since its purpose is BVH construction, not rasterization. Our main goal is to construct a BVH from a compressed hierarchical LOD representation of micro-poly geometry in real time. The quality of the hierarchical LOD representation is not crucial to demonstrate that this is possible and thus we do not compare our design decisions for preprocessing to the ones in Nanite directly. Quite possibly, the preprocessing in Nanite would give better results but that does not invalidate our approach for fast BVH build.

3. Preprocessing Phase

Our system utilizes a preprocessing phase with the same steps as Nanite. To keep this paper self-contained and to explain all the aspects that differ, we describe it in detail. It starts with the initial mesh clustering (Sec. 3.1), followed by merging and simplification of clusters to generate the LOD hierarchy (Sec. 3.2). Merging alone eventually yields too many boundary edges, so clusters are split when necessary (Sec. 3.3). Next, the cluster geometry is compressed lossily to reduce its memory footprint (Sec. 3.4) and stored in a GPU-friendly data structure (Sec. 3.5). As a scene typically consists of multiple geometric objects, all of these steps are performed for each object individually. Like Nanite, we assume that the original scene geometry consists of triangle meshes.

3.1. Initial Cluster Generation

In a first step triangles are converted into triangle pairs/quads. This is an efficient way to reduce memory consumption of the input data, as the quadification rate for micro-poly geometries is typically high. It is also well-aligned with the final ray-tracing hardware BVH layout that is going to be used (Sec. 4). Triangles which cannot be paired are stored as a degenerate quad, where the third and fourth vertex are the same. In the following, we refer to triangle pairs with a shared edge simply as quads.

Next, a binary BVH is built over all quads. For BVH construction, one can chose any high quality BVH build algorithm [MOB*21]. We opted for PLOC [MB18], as we use the same algorithm later in the cluster hierarchy construction phase

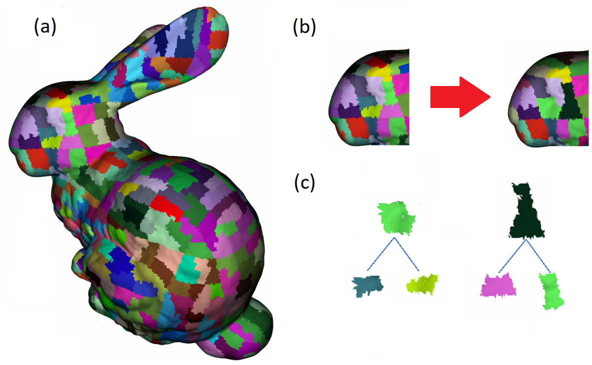


Figure 2: *(a) Initial generation of spatially coherent clusters with* ≤ 128 *quads (256 triangles) using a binary BVH. (b) Pairs of adjacent clusters sharing a common boundary are merged and simplified, while preserving the outer boundary. (c) The iterative merging of clusters generates a binary hierarchy over the clusters.*

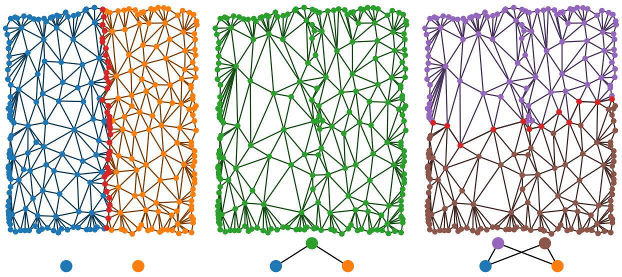
(Sec. 3.2). The resulting binary BVH is then traversed top-down and each subtree containing ≤ 128 quads (256 triangles) and ≤ 256 vertices is converted into a cluster. These constraints are due to the compression scheme applied later (Sec. 3.4). The cluster geometry itself is again represented as an indexed quad mesh. Relying on a BVH to extract the initial clusters efficiently identifies spatially coherent sets of quads (Fig. 2a). It is important to reduce the overlap between bounding boxes of neighboring clusters as much as possible, as a ray entering the overlapping region will have to descend into all associated cluster BVHs.

3.2. LOD Generation by Cluster Merging

Based on the set of initial clusters (Sec. 3.1), the goal is to generate a LOD hierarchy of clusters in a bottom-up manner. The initial clusters form the bottom-level or leaf level of the hierarchy, corresponding to the finest LOD resolution. Nodes higher up in the hierarchy contain simplified geometry of their descendants. The hierarchy is created by merging pairs of clusters while at the same time simplifying the merged geometry (Fig. 2b, 2c). It is important that the simplification process preserves the boundary edges of the merged cluster, such that adjacent clusters can select a different LOD without introducing visible cracks at the shared boundary.

For mesh simplification we rely on *MeshOptimizer* [Mes23]. It strives to maintain the overall appearance by preserving the topology of the original mesh including attributes like seams and boundaries. The output of the simplification step is a new vertex index buffer, which uses a subset of the vertices in the input index buffer.

Identifying pairs of clusters follows the same iterative logic as the PLOC algorithm [MB18] for building BVHs. It works by managing an array of active clusters which is initialized from the set of initial clusters. In each PLOC iteration, all active clusters scan (within a given search radius) their neighbors and evaluate a *distance* function. In our case, that is the surface area of the merged cluster AABBs. The neighbor with the smallest *distance* is marked as the nearest neighbor. If two clusters mutually agree on being their nearest neighbors, they will be merged and a new cluster is



(a) *Two clusters* **(b)** *Merged & simplified* **(c)** *Split*

Figure 3: *(a) Two clusters are selected to be merged. Shared boundary vertices between clusters are shown in red. (b) The merged cluster is simplified but boundary edges are preserved. Due to abundant boundary edges, the merged cluster has more than 256 triangles. (c) In this case, the simplified cluster is split into two new clusters. That introduces new, longer boundary edges. Both clusters resulting from the split become parents of the original two clusters. That turns the LOD hierarchy from a binary tree into a binary DAG.*

created. The new cluster becomes the parent of the two input clusters. Hence, the merging process builds a binary hierarchy over the clusters. The new *parent* cluster now replaces one of its children in the active cluster array, while the other child gets marked as invalid. Note that for the nearest neighbor search, each cluster will test only clusters sharing a boundary edge (hence their AABBs must overlap). Finally, a compaction step removes all invalid entries in the active cluster array and the process continues with the next iteration.

The PLOC algorithm typically continues until only a single entry in the active cluster array remains, corresponding to the binary BVH root node. At this point the entire binary BVH hierarchy has been created. When merging clusters, the assumption of a single root node does no longer hold, as the cluster merging process can fail. A failing merge can be caused by several factors, e.g. the merged cluster cannot be simplified enough (the reduced number of quads is not small enough) or the resulting cluster does not meet certain compression constraints (Sec. 3.4). That commonly happens because of too many boundary edges (Fig. 3b). A failed cluster merge is not considered critical. It only leads to having more cluster roots, which cannot be merged anymore, instead of having only one root node. The LOD selection process then works with multiple entry points in the LOD hierarchy (Sec. 4.1).

Basing the LOD hierarchy generation on a bottom-up BVH construction algorithm has the advantage that the hierarchy will be of high quality in terms of surface area, limiting the overlap between selected clusters in the hierarchy. That is beneficial when the selected cluster BVHs are fused together (Sec. 4.2).

3.3. Binary Directed Acyclic Graph (DAG2)

When merging and simplifying a pair of clusters, we aim for a 30-50% reduction in quad count, while preserving the boundary edges. If there are too many boundary edges, the reduction in the number of quads may be low (or none at all). The probability of these failed cluster merges increases towards the upper levels of the hierarchy (Fig. 3b), resulting in an unsatisfactory maximal simplification of the given geometry. Like Nanite, we reduce the probability of failed merges through splitting: If a merge fails, we take the merged cluster and cut it approximately in half while trying to keep the number of edges of the newly created boundary small (Fig. 3c). Next, for each half a new cluster is created with both input clusters as children. The two new clusters replace their two children in the active cluster array. In the next iteration of the PLOC-based merge process these newly created clusters will try to find different clusters to merge as we prohibit direct re-merging with the other half.

Cutting a merged cluster in a new way and reinserting both halves back in the cluster merging process requires a change of the LOD hierarchy representation, namely the transition from a binary tree (Fig. 3b) to a binary DAG (Fig. 3c), where nodes can have two parents instead of one. We refer to a binary DAG as *DAG2*. Special care needs to be taken when traversing the DAG2 for LOD cluster selection (Sec. 4.1 and App. A). Switching from a binary tree to a DAG2 representation allows us to get better cluster merging in the upper levels of the hierarchy, thereby reducing the number of cluster roots by up to $2\times$. Note that Nanite applies splitting in each step, uses different heuristics, and permits topological changes in simplification steps. This way, it achieves even better simplification.

3.4. Cluster Compression

Given a hierarchy of clusters (Sec. 3.3), we compress each cluster to reduce the overall memory footprint. First, all vertices are quantized with respect to the bounding box of the geometric object the cluster belongs to using *16 bits* per dimension. That means a 3D vertex will require $3 \times 2 = 6$ bytes after quantization instead of the 12 bytes needed with 32-bit single precision floating point values. We limit the maximum number of vertices to 256, which allows for using *8-bit* indices for the indexed quad mesh of the cluster. The maximum number of quads inside a cluster is set to 128 which corresponds to 256 triangles. Also the quads inside the cluster are ordered spatially by building a small binary BVH over them and rearranging the quads based on a depth-first BVH traversal. The spatial ordering helps to increase the BVH quality over the quads inside a cluster (Sec. 4.2).

The vertex quantization with respect to the bounding box of the geometric object guarantees vertex consistency across boundary edges between all clusters of the object, which is important as neighboring clusters can be subdivided to different LODs. Identical vertices in different clusters are affected by quantization error in the same way, such that water tightness is preserved. In terms of memory consumption our simple vertex quantization and indexing scheme reduces average memory consumption to 8-12 bytes per quad / 4-6 bytes per triangle (Tab. 1). That allows for storing 165-222 million triangles within a gigabyte of memory. Note that shared vertices at the cluster boundary are stored in all adjacent clusters. This vertex replication increases the memory consumption of the LOD hierarchy.

Table 1: *Our lossy cluster compression scheme reduces memory consumption by $\sim 2.5-2.9\times$ when comparing the finest cluster LOD level to the uncompressed triangle/grid meshes. On average the scheme reduces memory consumption per triangle to $\sim 4.8-6.5$ bytes. Including LOD data for all levels, which includes vertex replication in clusters, the memory consumption is still $\sim 1.2-1.5\times$ lower than the standard uncompressed mesh/grid case. For the grid case, all LOD levels are generated on the fly from the original grid. Hence, no explicit storage of simplified clusters is required.*

	Thai (mesh)	Rungholt (mesh)	Landscape (grid)
# Triangles	200 M	100 M	134 M
# Vertices	30 M	44.3 M	67.1 M
	Uncompressed triangle meshes/grid		
Memory	2.94 GB	1.57 GB	0.77 GB
Bytes per triangle	15.6	17.7	6.0
	Compressed clusters (finest LOD only)		
# Clusters	1.19 M	0.56 M	1.01 M
Memory	0.9 GB	0.57 GB	0.29 GB
Bytes per triangle	4.8	6.5	2.3
	Compressed clusters (all LODs)		
# Clusters	2.45 M	1.21 M	1.01 M
Memory	1.97 GB	1.28 GB	0.29 GB
Bytes per triangle	5.7	6.6	2.3
	Cluster roots (coarsest LOD only)		
# Clusters	16.6 k	192.6 k	65 k
# Triangles	2.7 M	21.8 M	8.3 M
Max simplification	71.8×	4.2×	16×

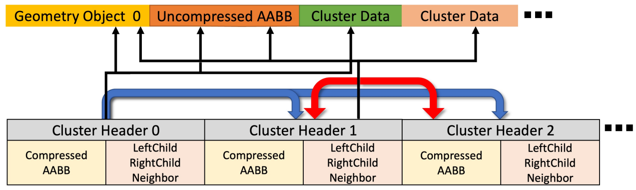


Figure 4: *Each compressed geometric object holds an uncompressed AABB which is used for decompressing all compressed vertices of all clusters belonging to the object. The compressed vertex and quad index data vary in size depending on the number of quads per cluster and the amount of vertex sharing. For each cluster a* cluster header *is created which contains the cluster's compressed AABB, used during LOD selection, as well as references to the geometric object's uncompressed AABB (black), the cluster's compressed data (black), the two* cluster header *children (blue) and the neighbor (red) used for the DAG2 representation.*

3.5. Cluster Data Structures

As cluster compression (Sec. 3.4) relies on the geometric object the cluster belongs to, all compressed vertex and quad data of all clusters belonging to the geometric object are stored in a consecutive region of memory (Fig. 4). The amount of data per cluster will vary depending on the number of quads per cluster and the amount of vertex sharing.

The LOD selection phase (Sec. 4.1) does not access the compressed vertex and quad data, so we separate the data required by this phase from the compressed cluster data and store it in a separate 40-byte *cluster header* (Fig. 4). The *cluster header* contains the compressed bounding box of the cluster, adjacency information like references to the left and right child, as well as a reference to a potential neighbor cluster, which arose from a split (Sec. 3.3). It also contains a reference to the AABB of the geometric object as well as an offset which marks the beginning of cluster data within the compressed cluster data of the object.

4. Per-Frame Phase

The per-frame phase builds a BVH containing a selection of clusters with appropriate LOD for the current frame. Such a BLAS could be reused in a subsequent frame when the LOD selection has not changed, but we evaluate using the worst case. It produces this BVH directly in the format required by our target GPU (an Intel Arc A770). The GPU's ray tracing units require a 6-wide quantized/compressed BVH with quads at leaf level [Int23]. We refer to such a BVH as *QBVH6*. Each QBVH6 node and quad leaf take 64 bytes. After selecting clusters (Sec. 4.1), we perform decompression and write a small QBVH6 for each selected cluster to memory (Sec. 4.2). Then we fuse these QBVH6 into a full BLAS by building the upper levels (Sec. 4.3).

4.1. LOD Cluster Selection

Given a DAG2 hierarchy of clusters (Sec. 3.3) a subset of clusters needs to be selected which will be decompressed and converted into a QBVH6 (Sec. 4.2). We therefore traverse the DAG2 in a top-down manner, starting at the root clusters, which represent the coarsest LOD. Each node gets to decide whether its LOD is sufficient based on the compressed AABB stored in the cluster header. A node and its neighbor arising from the same split (if any) always store the same AABB and thus they make the same LOD decision. If the LOD is sufficient, the cluster is selected for inclusion in the BVH. Otherwise, all of its children will be tested subsequently. In Appendix A, we prove that such a top-down traversal is guaranteed to give a complete and crack-free mesh.

The heuristic that determines whether a cluster has a sufficient LOD differentiates between clusters inside and outside the view frustum. For the sake of secondary rays, clusters completely outside the view frustum are not discarded. However, they use a coarser LOD, solely based on the distance to the viewer. For clusters inside the view frustum, the cluster's compressed AABB (stored in the cluster header) is projected onto the image plane. Then the length of the diagonal of the projection of the 2D AABB is computed. The DAG2 top-down traversal stops if the diagonal length is smaller than a threshold, in our case 24 pixels. Note that LOD selection happens once per frame, not per ray, so these heuristics apply equally to all primary and secondary rays.

The cluster selection is implemented in two steps: First during top-down traversal, we mark all clusters selected by the LOD heuristic. In the second step, all marked clusters are added to the list of active clusters per frame. The two step approach is necessary to avoid adding the same cluster multiple times during top-down traversal, as a cluster in a DAG2 can have two parents.

Nanite [KSW21] instead relies on a monotonic heuristic, where each cluster that has sufficient LOD but a parent with insufficient LOD must be rendered. An additional tree data structure is used to cull irrelevant clusters early. That enables greater parallelism but requires another data structure and makes it harder to design a good LOD heuristic.

4.2. Cluster Decompression and BVH Build

Once all currently active clusters per frame are selected (Sec. 4.1), references to these clusters are passed to a GPU BVH builder based on PLOC++ [BDTD22]. Our implementation of this and the next step is hardware specific as it exploits knowledge of the hardware BVH layout. Support for other hardware requires alternative implementations. The builder first iterates over the list of active clusters and decompresses the cluster's compressed vertices into the GPU's shared local memory. This does not consume main memory bandwidth, as shared local memory is not backed up by the GPU's cache/memory hierarchy. As we limit the number of quads per cluster to ≤ 128 and let a sub-group (wave) work on a single cluster, enough shared local memory space per sub-group is available to hold all decompressed vertices.

The next step builds a small QBVH6 over the decompressed data. First, four decompressed vertices per quad are directly converted into the QBVH6's quad leaf layout and stored out to global memory. Next, the decompressed vertex data per quad in shared local memory are overwritten with the quad's AABB, as the vertex data are no longer needed. Based on the list of AABBs the QBVH6, is now built bottom-up in an iterative manner. In each iteration, groups of six AABBs are assembled together in a dense fashion and a new QBVH6 node per assembled group is stored out to global memory. As the quads per cluster are spatially ordered (Sec. 3.4), the overlap between the densely packed nodes is limited and therefore the quality of the resulting QBVH6 remains high.

Note that we experimented with more sophisticated QBVH6 build algorithms over the cluster's quads but due to the limited number of quads, requiring only a few inner QBVH6 nodes, the quality and therefore ray tracing performance impact was very limited. Either way, the AABB of a quad is tested before a ray-quad intersection test. Thus, the number of ray-quad intersection tests is mostly unaffected. In the worst case, a lower quality of the QBVH6 for a single cluster manifests in a few additional traversal steps, which is typically negligible. Also with a less dense packing, more inner QBVH6 nodes have to be written out to global memory which is costly (Sec. 5). Once the root node of the cluster's QBVH6 is created, its uncompressed AABB as well as a reference to the QBVH6 is also written out to global memory. These data are required by the next phase, which fuses the cluster BVHs together (Sec. 4.3). In terms of memory consumption, the maximum size of a cluster with 128 quads and 256 vertices is 2.0 kB, converting to 9.7 kB of QBVH6 data, which is a $\sim 5\times$ size expansion.

4.3. Cluster BVH Fusing

The uncompressed AABBs of all cluster QBVH6 root nodes are the input of the modified PLOC++ builder, which builds a QBVH6 over them. Its final phase connects the leaf nodes of this QBVH6 directly to the previously written root nodes of the cluster QBVH6s. This *fuses* all cluster QBVH6s together into a single QBVH6, which can be used to efficiently ray trace all decompressed geometry in the scene.

Note that instead of a full QBVH6 build over cluster AABBs one could have relied on a refitting-based approach, as the cluster positions with respect to the scene stay mostly constant. This would likely reduce fusing cost. However, extracting a refitable BVH out of the DAG2 structure will also introduce an overhead. A full BVH rebuild has the additional advantage of offering more flexibility, e.g. other types of primitives, besides the cluster root nodes, can be easily added to the list of primitives for the final QBVH6 build.

Another alternative is to prepare one BLAS per cluster and to replace our cluster BVH fusing by a standard TLAS build. However, we expect to have more compressed clusters loaded into VRAM than are actually used in one frame. Since a QBVH6 is $\sim 5\times$ bigger than a compressed cluster, this approach utilizes considerably more memory.

5. Results

For our evaluation, we implemented our approach on top of Embree 4 [Int23] using oneAPI's SYCL/DPC++ [Int21] as GPU programming language. Embree 4 supports ray queries similar to those in DXR or Vulkan. Our hardware platform uses an Intel Arc A770 as GPU (32 Xe cores, 16 GB GDDR6 memory, 256-bit memory interface), and a Core i7-11850H as host CPU running Ubuntu 22.04 Linux. All kernel timings in this section are measured on the GPU such that kernel launch overhead is excluded. On average, the QBVH6 format required by the GPU's ray tracing cores takes up a gigabyte of memory for ~24 M triangles.

Embree 4 has been chosen because its open source code directly exposes the hardware-specific BVH layout, which is currently not possible with DXR or Vulkan. We extended it to support compressed clusters as another primitive type and their decompression and conversion into the QBVH6 layout (Sec. 4.2) has been integrated directly into Embree 4's GPU BVH builder. All cluster preprocessing is done on the CPU while all per-frame phases are executed as GPU compute kernels: DAG2-based LOD selection (Sec. 4.1), cluster decompression with cluster QBVH6 build (Sec. 4.2), and BVH fusing (Sec. 4.3).

Table 2: *Cost breakdown in ms for the three per-frame phases. The total per-frame overhead is $\sim 5.7 - 9$ ms. The cluster decompression and per-cluster QBVH6 build takes longest. Combined with fusing of cluster QBVH6s, the two phases together build a QBVH6 over all triangles in the selected clusters, reaching a QBVH6 build performance of $\geq$4 GTriangles/s which is more than $10\times$ higher compared to building a QBVH6 over all triangles in the selected clusters (without utilizing the clustering).*

	Thai	Rungholt	Landscape
LOD selection	1.7	0.9	1.2
Decompression+QBVH6	2.4	5.9	2.6
Fusing cluster QBVH6s	1.6	2.3	2.0
Total	5.7	9.0	5.8
QBVH6 build perf. [GTris/s]	4.0	4.8	4.5

Table 3: *Per-frame statistics for cluster decompression and QBVH6 construction (for views in Figure 1). For the Thai Statues, the LOD phase has selected ∼104 k clusters, which correspond to ∼16 M triangles. Decompression and cluster QBVH6 construction take 2.4 ms in total. That corresponds to ∼ 75% of the attainable memory copy performance on the Intel Arc A770 (∼410 GB/s).*

	# Clusters	# Triangles	Cluster QBVH6	Time	Read mem.	Write mem.	Total mem.	vs. MemCopy
Thai Statues	104 k	16 M	0.62 GB	2.4 ms	45.4 GB/s	260.2 GB/s	305.6 GB/s	74.5%
Rungholt	232 k	40 M	1.56 GB	5.9 ms	44.8 GB/s	264.0 GB/s	309.2 GB/s	75.4%
Landscape	174 k	21 M	0.82 GB	2.6 ms	20.3 GB/s	316.9 GB/s	337.2 GB/s	82.2%

We present timings for these per-frame phases (Sec. 5.1) and for rendering (Sec. 5.2). Then we assess the visual quality of our LOD mechanism (Sec. 5.3). We also apply our approach to (animated) grids with known topology (Sec. 5.4). Finally, we discuss limitations (Sec. 5.5).

5.1. Timings for the Per-Frame Phases

The first per-frame phase is the LOD selection (Sec. 4.1). The cost of the two step approach is around 0.9-1.7 ms (Tab. 2). The result of this phase is a subset of clusters which will be decompressed and converted into the QBVH6 layout in the next phase. Tab. 3 shows the number of selected clusters, the number of triangles contained in these clusters, and the total accumulated size of all QBVH6s for clusters after being written out to memory. Loading of cluster data, decompression, QBVH6 construction, and writing out ∼0.6-1.6 GB of QBVH6 data to memory takes 2.4-5.9 ms. In comparison to the attainable memory copy performance of the target GPU (410 GB/s), our approach reaches 305-337 GB/s, which corresponds to 74-82%.

The final per-frame phase of fusing cluster QBVH6s takes 1.6-2.3 ms, i.e. slightly longer than the LOD selection. Comparing the costs of all three phases (Tab. 2), the cluster decompression with QBVH6 construction takes longest because of its high memory bandwidth requirements. The total per-frame overhead of all three phases combined is 5.7-9 ms.

The two latter per-frame phases are equivalent to a full QBVH6 GPU build for all geometry selected in this frame. However, the two-phase approach with preprocessing is way more efficient than building the QBVH6 over triangles/quads. For example, the *Thai Statues* scene has a combined cost of 4 ms. For 16 M triangles, that corresponds to a GPU BVH build performance of ∼ 4 GTriangles/s (Tab. 2). Compared to a regular GPU BVH build for triangles/quads this throughput is more than 10 times higher. The higher throughput is due to using a pre-defined hierarchy (clusters) to quickly build the lowest levels of the QBVH6, which is the most costly part of BVH construction [HMF07a,HMF07b]. In addition to being significantly faster, building a BVH over clusters consumes less memory bandwidth and memory capacity during the QBVH6 build.

5.2. Timings for Ray Tracing and Path Tracing

The per-frame overhead and memory consumption of our approach depend directly on the number of selected clusters per frame (Fig. 5). The LOD selection, needs to be very aggressive to keep the overhead small. If the maximum LOD simplification per geometric object is limited, the LOD selection will often select too fine LODs, especially for geometry far away from the viewer. This will increase the per-frame overhead.

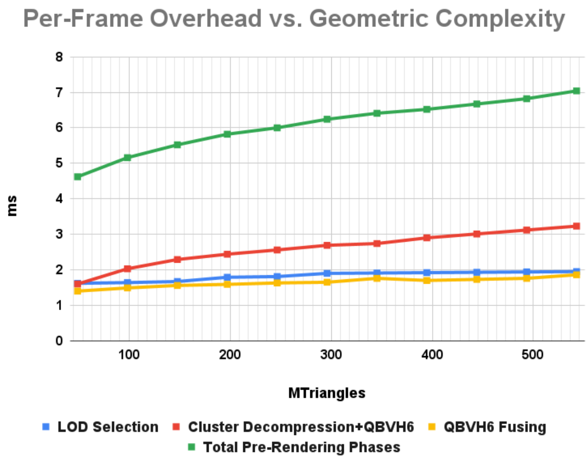


Figure 5: *Per-frame overhead vs. geometric complexity introduced by the three per-frame phases for an increasing number of Thai Statues. From 65 M to 542 M triangles the overhead increases linearly with scene complexity. The cluster decompression with QBVH6 build is the most costly phase.*

Due to the LOD selection, only a fraction of the total geometry will end up in the per-frame QBVH6. Tab. 4 shows that the per-frame QBVH6 is $2.5-13.4\times$ smaller than the QBVH6 built over the original geometry. A larger QBVH6 size has a significant impact on the ray tracing performance due to additional ray traversal/intersection steps and a higher number of cache misses. Tab. 4 shows that primary ray performance is $1.5-2.8\times$ slower than for the per-frame QBVH6. The ratio changes for diffuse 1-bounce path tracing to $1.3-1.7\times$ as only half of the time is spent tracing rays and due to cache/memory effects caused by less coherent rays.

5.3. Visual Quality of our Hierarchical LOD

Fig. 6 shows to what extent our LOD mechanism manages to provide similar quality on surfaces with different LOD. Since the LOD hierarchy has the original mesh at its leaf-level, nearby surfaces are rendered with high fidelity. On more distant surfaces, we observe slight artifacts. These occur partly because we do not store per-vertex normals and instead rely on triangle normal vectors for shading. In addition, some of the triangles at low LOD are nearly as large as their corresponding clusters and thus they cover more pixels compared to triangles of a higher LOD. Since all clusters have similar quad counts, that indicates a greater number of boundary edges in clusters of low LOD. A possible remedy is to use splitting (Sec. 3.3) more frequently, like Nanite. Under camera mo-

Table 4: *Rendering performance comparison* (1920 × 1080, *1 spp) of our approach against a QBVH6 built over triangles at the finest LOD. Since the final QBVH6 is significantly smaller, our approach accesses less data during ray traversal achieving a* $1.5-2.8\times$ *faster rendering performance for primary visibility (PV), which is dominated by tracing rays. For diffuse path tracing (PT) where only 50% of the time is spent tracing rays, and memory effects impact performance even more, the speedup is reduced to* $1.3-1.7\times$*. The cluster-based hierarchy inside our QBVH6 moderately affects SAH quality, as our QBVH6 has a* $\sim 1.2\times$ *higher SAH than a QBVH6 built over all cluster triangles.*

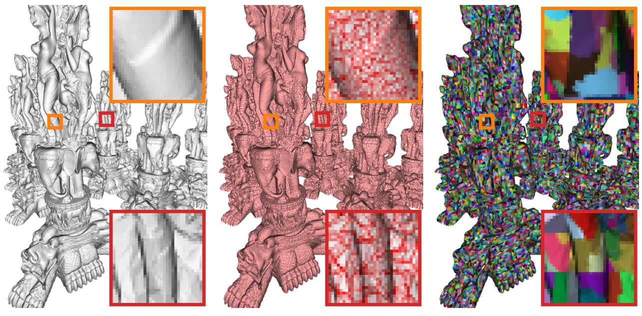
	Our approach						Finest LOD			
	# Tris	QBVH6	SAH	Per-frame	PV	PT	# Tris	QBVH6	PV	PT
Thai Statues	16M	0.62 GB	1.17×	5.7 ms	1.0 ms	24.4 ms	200 M	8.33 GB	2.8 ms (2.8×)	42.1 ms (1.7×)
Rungholt	40M	1.56 GB	1.19×	9.0 ms	1.0 ms	25.1 ms	100 M	3.92 GB	1.6 ms (1.6×)	32.6 ms (1.3×)
Landscape	21M	0.82 GB	1.20×	5.8 ms	1.2 ms	24.4 ms	134 M	5.66 GB	1.8 ms (1.5×)	31.7 ms (1.3×)

tion, these issues manifest as minor popping artifacts, which can be seen in our supplemental video. On the other hand, the LOD selection (Sec. 4.1) successfully selects clusters of similar size in screen space, even on distant surfaces.

5.4. Alternative LOD Hierarchies and Dynamic Geometry

Our approach is not limited to compressed cluster meshes only but supports other types of input formats as well, e.g. quad-tree based LOD hierarchies over grid-like structures as typically used for representing large terrains (Fig. 1 right). The only major difference compared to our compressed, clustered meshes is that clusters at each LOD level can be extracted from the grid structure directly. No cluster merging is required. Besides, there is no need to store index buffers. For our implementation, we use a bilinear patch-based compression scheme [BVW21], which is suited for grid-like topology. It reduces the storage cost to ~ 2.3 bytes per triangle.

The grid-like structure allows us to exploit another advantage of our approach: As the cluster's compressed vertex data are decompressed into shared local memory (Sec. 4.2), we can modify the vertices before converting them to the QBVH6 layout without additional memory traffic. This allows us to implement dynamic crack



(a) *Shaded scene* **(b)** *Tesselation* **(c)** *Clusters*

Figure 6: *(a) 12 Thai Statues rendered using our LOD mechanism. On nearby geometry (orange inset), results are detailed. The same surface seen at a greater distance (red inset) shows minor artifacts. (b) A red wireframe view reveals that many triangles at the lower LOD (red inset) are actually larger in screen space than triangles closer to the camera (orange inset). (c) Visualizing different clusters in different colors shows that clusters on more distant geometry are slightly smaller in screen space, unlike some of the triangles.*

fixing at cluster boundaries (due to different LODs) and to blend vertices between different LOD levels to support continuous LOD. Our supplemental video shows the Landscape scene with continuous LOD.

A similar approach also works for meshes with animated control points. Fig. 7 shows an animated model consisting of 52 k patches (B-spline and Gregory patches). LOD heuristics select a suitable tesselation level for each of the four (shared) patch edges. The patch interior is tessellated according to the maximum of the four edge levels. Then the tesselated patch is stored as lossy compressed cluster with header (with up to 128 quads). At this point, our usual per-frame processing generates a BLAS.

5.5. Limitations

Our system is a proof of concept focusing on fast BVH build and compared to more mature technologies like Nanite, it has clear shortcomings. At low LOD, our method sometimes yields too large triangles on curved surfaces (Sec. 5.3). We also cannot simplify meshes to an arbitrary extent (Tab. 1). A more sophisticated preprocessing phase, possibly coupled with a wider DAG, might overcome these problems. There is no streaming mechanism but it could be added naturally. Efficient occlusion culling is more challenging. For path tracing, we do not want to cull anything but we would like to use a lower LOD for occluded surfaces. Nanite rasterizes visible clusters from the previous frame into a hierarchical depth buffer [KSW21]. It is not obvious how to do something similar for ray tracing without building a BVH twice per frame.

For long diagonal geometry, e.g. branches of a tree, our dense QBVH6 generation per cluster is rather inefficient in terms of BVH quality and therefore ray tracing performance. The impact could be largely mitigated by pre-splitting of long diagonal geometry before partitioning it into clusters. An alternative is to perform the spatial splits directly after decompression and to store out a larger QBVH6 to memory. However, this would increase the complexity of the decompression kernel and lead to slower run-time.

Vertices are quantized with respect to the bounding box of their geometric object (e.g. a single Thai statue). That ensures water tightness for all clusters of a given geometric object but not across different objects. It will also suffer from precision issues for objects with a large spatial extent due to 16bit quantization per dimension. The approach of Segovia et al. [SE10] could mitigate these issues by aligning bounding boxes of geometric objects to a global grid over the scene.

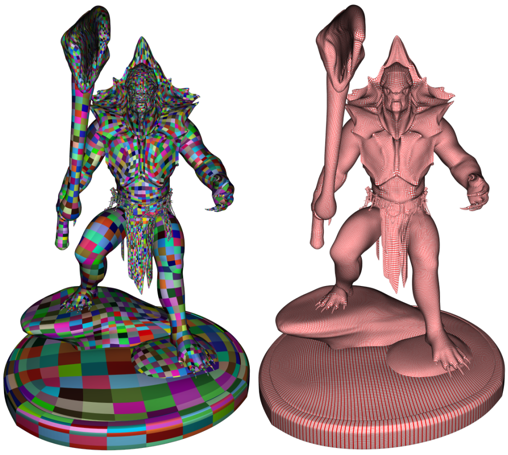


Figure 7: *An animated model consisting of 52 k patches. It is rendered with LOD by converting each patch into a compressed cluster with the appropriate LOD for one frame. In this view, the 52 k patches are tessellated into 1.01 M triangles. LOD selection, tessellation (including crack-fixing) and cluster conversion take ∼0.5 ms.*

6. Conclusions

Starting the BVH construction with small spatially coherent clusters of quads is ten times faster than constructing a BVH over individual triangles. This immense speedup is made possible by a preprocessing phase with clustering and compression. Arguably, per-mesh preprocessing like that is acceptable in many cases. We have demonstrated that this approach is useful for hierarchical level of detail. Our proof of concept still has many shortcomings but it shows that "Nanite with ray tracing" is more feasible than one might think. The per-frame overhead of ≤ 9 ms is greater than the cost of rasterizing a similar scene with Nanite, largely because writing a BVH to memory takes a lot of bandwidth, but it is not that far from being practical. All of this is enabled by the fact that Embree 4 comes with an open-source BVH builder. Turning our technique into a cross-vendor solution requires a standardization effort.

7. Future Work

There is room for improvement in our preprocessing phase (Sec. 5.5). It would be interesting to feed our method directly with clusters produced by Nanite. Nanite does not optimize for AABBs with small surface area but its preprocessing may give better results for ray tracing nonetheless. A mechanism similar to sampler feedback for sparse virtual texturing but based on ray differentials might be an adequate replacement for occlusion culling. We already support dynamic, patch-based geometry (Sec. 5.4). To support skinned triangle meshes, the main challenge is LOD selection.

From a general API perspective, we would like to investigate how to extend the current DXR and Vulkan APIs to support our approach. For that a standardized format for compressed geometry clusters would be valuable. Applications could convert their own geometry representation to this compressed format (either offline or at run time). The DXR and Vulkan implementation in the driver could decompress these geometries before building the BLAS (Secs. 4.2 and 4.3). That places the interface at a point where memory bandwidth requirements are low and flexibility is high. Code written for such an API remains useful when the hardware BVH layout changes. The main bottleneck of our method is the bandwidth required to write cluster QBVH6s to VRAM. Thus, we hope that future hardware will utilize more compact BVH layouts. An API extension which allows the user to fuse multiple BLAS together into a single BLAS (Sec. 4.3), without the usual TLAS/BLAS separation, could also be valuable.

Acknowledgements

The Thai Statue is courtesy of the Stanford Computer Graphics Laboratory, Rungholt is courtesy of *kescha*, the Landscape scene is courtesy of Bartosz Domiczek, and the Barbarian model is courtesy of Autodesk (Jesse Sandifer is the original artist).

Appendix A: Characterizing Valid Cluster Selections

The clusters which will be rendered by Nanite are defined by a "view dependent cut of the DAG." From the presentation [KSW21], it is not entirely clear what requirements such a cut has to satisfy and why that guarantees complete and crack-free geometry. It took some effort to understand the details. In hopes that it will be useful to others, this appendix explains our learnings and provides a formal proof that the approach actually works as desired. We start with some basic definitions about DAGs.

Definition 1. A *DAG* has a finite, non-empty set of nodes V connected by directed edges $E \subset V \times V$ without cycles. In our context, each node is a cluster of geometry. For a node $v \in V$, consider its

$$\begin{aligned} \textit{parents} \ \ P(v) &:= \{p \in V \mid (v,p) \in E\}, \\ \textit{children} \ \ C(v) &:= \{c \in V \mid (c,v) \in E\}, \\ \textit{siblings} \ \ S(v) &:= \{s \in V \mid P(v) \cap P(s) \neq \emptyset \ \text{ or } \ v = s\}. \end{aligned}$$

That means two siblings have at least one parent in common and nodes qualify as their own sibling. We call the DAG an *LOD DAG* if siblings have all parents in common, i.e. for all $v \in V$ and $s \in S(v)$, $P(v) = P(s)$ (Fig. 8). *Leaves* are nodes with no children, *roots* are nodes with no parents.

In our (and Nanite's) hierarchical LOD DAG, the combined geometry for all leaves is the original mesh. By definition, the original mesh is complete and crack-free. Now consider a cluster $w \in V$ that is not a root. Its siblings $S(w)$ are clusters that have been grouped with w. The result of merging, simplifying and splitting these clusters are the parents $P(w)$ (Fig. 3c). Thus, the siblings $S(w)$ describe the same patch of geometry as the parents $P(w)$ and the geometry has the same boundary edges. If we have a complete and crack-free selection of clusters $G \subseteq V$ with $w \in G$ and $S(w) \subseteq G$, removing siblings of w and adding parents of w gives us another complete and crack-free selection, but with lower level of detail.

That is a full description of what we can do without introducing problems but it is hard to reason about. Given a selection of clusters $G \subseteq V$, we have no good way to tell whether it is complete and crack-free. In the following, we provide such a characterization using elementary graph theoretic terms.

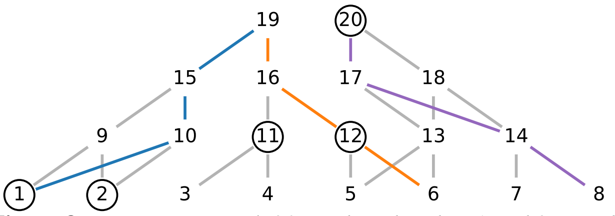


Figure 8: *A LOD DAG with 20 numbered nodes. 1 and 2 are siblings. Their parents are 9 and 10. 3 and 4 are children of 11. The circled nodes form a cut set G combining three different LODs. Three paths are shown in different colors and each of them meets this cut set exactly once. 12 and 13 arose through splitting but only one of them is selected for the cut set. Thus, boundary edges created by splitting separate clusters of different LOD.*

Definition 2. A *path* through the DAG V is a sequence of nodes $v_1,\dots,v_L \in V$ such that v_1 is a leaf, v_L is a root and for all $i \in \{1,\dots,L-1\}$, v_{i+1} is a parent of v_i. We call $G \subseteq V$ a *cut set* if each path through the DAG meets exactly one node in G.

Cut sets can be thought of as a barrier between (or at) leaves and roots that you encounter exactly once, no matter what path along the edges you choose (Fig. 8). We will learn, that cut sets are exactly the selections of clusters that yield complete and crack-free geometry. First, we prove that the unproblematic operations identified above yield cut sets.

Proposition 1. The set of all leaves in an LOD DAG V is a cut set. Let $T_n \subseteq V$ be a cut set and let $w_n \in T_n$ be a node that is not a root with $S(w_n) \subseteq T_n$. Then

$$T_{n+1} := (T_n \setminus S(w_n)) \cup P(w_n) \tag{1}$$

is also a cut set.

Proof Each path contains exactly one leaf, so the set of all leaves is a cut set. An arbitrary path $v_1,\dots,v_L \in V$ has exactly one node $v_i \in T_n$. If $v_i \in S(w_n)$, $v_i \notin T_{n+1}$ but

$$v_{i+1} \in P(v_i) = P(w_n) \subseteq T_{n+1}.$$

If $v_i \notin S(w_n)$, $v_i \in T_{n+1}$ but since the path does not meet $S(w_n) \subseteq T_n$, it does not meet $P(w_n)$ either. In both cases, the path meets T_{n+1} exactly once, i.e. T_{n+1} is a cut set. □

Now we go the other way and show that each cut set can be constructed with these unproblematic operations.

Proposition 2. Let $G \subseteq V$ be a cut set in an LOD DAG V. Then G can be constructed from the set of leaves by replacing siblings by their parents repeatedly, as in Eq. 1.

Proof We will prove that the following algorithm terminates with $T_n = G$:

Set T_1 to the set of all leaves in V.
For $n = 1,2,\dots$ until there is no $w_n \in T_n \setminus G$ with $S(w_n) \subseteq T_n$:
  Pick a $w_n \in T_n \setminus G$ with $S(w_n) \subseteq T_n$.
  $T_{n+1} := (T_n \setminus S(w_n)) \cup P(w_n)$.

We prove by induction over n, that T_n is always a cut set and that no path meets a node in G before a node in T_n.

Induction start, $n = 1$: The set of all leaves T_1 is a cut set (Prop. 1). Paths begin at leaves, so they meet T_1 first.
Induction step, $n \to n+1$: By the induction hypothesis, a path that meets $w_n \in T_n \setminus G$ meets G afterwards, so w_n is not a root. Applying Prop. 1, we find that T_{n+1} is a cut set.
An arbitrary path $v_1,\dots,v_L \in V$ meets the cut sets T_n, T_{n+1}, G at v_i, v_j, v_k, respectively. We have to prove $j \leq k$. By the induction hypothesis, $i \leq k$. If $v_i \notin S(w_n)$, $v_j = v_i \in T_{n+1}$ and hence $j \leq k$. Now consider the case $v_i \in S(w_n)$. Let $u_1,\dots,u_K \in V$ be a path that meets $w_n = u_l$ at an index $l \in \{1,\dots,K\}$. It meets G after $w_n \in T_n \setminus G$. Then $v_1,\dots,v_i,u_{l+1},\dots,u_K$ is another path and it meets G after v_i. Therefore, $v_i \notin G$ and thus $k \geq i+1$. And since $v_{i+1} \in P(v_i) = P(w_n)$, $i+1 = j$. That completes the induction.
The algorithm must terminate because the graph is finite and acyclic and in each step, at least one node (namely w_n) is replaced by its parents. If the loop ran long enough, it would eventually reach roots and after additional steps T_n would be the empty set.
Now consider the situation upon termination. An arbitrary path $v_1,\dots,v_L \in V$ meets the cut sets T_n, G at v_i, v_k, respectively. We have proven $i \leq k$. Assume that the path has been chosen such that $k-i$ is maximal.
Suppose $i < k$. Then $v_i \in T_n \setminus G$ must have a sibling $s \in S(v_i)$ with $s \notin T_n$. Let $u_1,\dots,u_K \in V$ be a path that meets $s = u_l$ at an index $l \in \{1,\dots,K\}$. Then $u_1,\dots,u_l,v_{i+1},\dots,v_L$ is another path. Since $v_i \in T_n$, $v_{i+1},\dots,v_L \notin T_n$ and therefore $u_j \in T_n$ for some $j < l$. The index where the composited path meets G is $k+l-i$. But that means

$$(k+l-i)-j = k+(l-j)-i > k-i,$$

which contradicts $k-i$ being maximal. Thus, $i = k$ for all paths and therefore $T_n = G$. □

Now we know that complete and crack-free selections of clusters correspond to cut sets of the LOD DAG. Furthermore, we know that we can construct them in a bottom-up fashion starting at the leaves. However, our cluster selection works top-down. Thankfully, the same proof works for that case:

Corollary 1. Let $G \subseteq V$ be a cut set in an LOD DAG. Then G can be constructed from the set of roots by replacing parents and the parents of their children by their children repeatedly.

Proof Reverse the direction of each edge of the DAG. That changes nothing about which sets are cut sets and the graph is still an LOD DAG afterwards. However, it turns roots into leaves, parents into children and vice versa. □

References

[BDTD22] Benthin C., Drabinski R., Tessari L., Dittebrandt A.: PLOC++: Parallel locally-ordered clustering for bounding volume hierarchy construction revisited. *Proc. ACM Comput. Graph. Interact. Tech. 5*, 3 (2022). doi:10.1145/3543867. 1, 6

[BM23] Bickford N., Moreton H.: Getting started with compressed micro-meshes. In *NVIDIA GPU Technology Conference* (Mar. 2023). URL: https://register.nvidia.com/flow/nvidia/gtcspring2023/attendeeportal/page/sessioncatalog/session/1666430278669001BFSR. 2

[BVW21] BENTHIN C., VAIDYANATHAN K., WOOP S.: Ray tracing lossy compressed grid primitives. In *Eurographics 2021 - Short Papers* (2021), Theisel H., Wimmer M., (Eds.). doi:10.2312/egs.20211009. 2, 8

[BWN*15] BENTHIN C., WOOP S., NIESSNER M., SELGRAD K., WALD I.: Efficient ray tracing of subdivision surfaces using tessellation caching. In *Proceedings of High-Performance Graphics* (2015), ACM. doi:10.1145/2790060.2790061. 2

[DHW*11] DJEU P., HUNT W., WANG R., ELHASSAN I., STOLL G., MARK W. R.: Razor: An architecture for dynamic multiresolution ray tracing. *ACM Trans. Graph. 30*, 5 (2011). doi:10.1145/2019627.2019634. 2

[HMF07a] HUNT W., MARK W., FUSSELL D.: Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In *Proceedings of Symposium on Interactive Ray Tracing* (2007), pp. 47–54. 7

[HMF07b] HUNT W., MARK W., FUSSELL D.: Fast and Lazy Build of Acceleration Structures from Scene Hierarchies. In *IEEE Symposium on Interactive Ray Tracing* (2007). 7

[IKH22] IKEDA S., KULKARNI P., HARADA T.: *Multi-Resolution Geometric Representation using Bounding Volume Hierarchy for Ray Tracing*. Tech. rep., Advanced Micro Devices, Inc., 2022. URL: https://gpuopen.com/download/publications/GPUOpen_BVHApproximation.pdf. 2

[Int21] INTEL CORPORATION: oneAPI programming model, 2021. URL: https://www.oneapi.com/. 6

[Int23] INTEL CORPORATION: Embree 4.0, 2023. URL: https://github.com/embree/embree. 2, 5, 6

[Khr20] KHRONOS GROUP: Vulkan Ray Tracing Extensions Specification, 2020. URL: https://www.khronos.org/registry/vulkan/specs/1.2-extensions/man/html/VK_KHR_ray_tracing.html. 1

[KSW21] KARIS B., STUBBE R., WIHLIDAL G.: A deep dive into Nanite virtualized geometry, 2021. in Advances in Real-Time Rendering in Games: Part I (proc. SIGGRAPH courses). URL: https://advances.realtimerendering.com/s2021/Karis_Nanite_SIGGRAPH_Advances_2021_final.pdf. 1, 2, 6, 8, 9

[MB18] MEISTER D., BITTNER J.: Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 24*, 3 (2018). doi:10.1109/TVCG.2017.2669983. 3

[Mes23] Mesh optimizer, 2023. URL: https://github.com/zeux/meshoptimizer. 3

[Mic20] MICROSOFT: DirectX Raytracing (DXR) Functional Spec, 2020. URL: https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html. 1

[MOB*21] MEISTER D., OGAKI S., BENTHIN C., DOYLE M. J., GUTHE M., BITTNER J.: A Survey on Bounding Volume Hierarchies for Ray Tracing. *Computer Graphics Forum 40*, 2 (2021). doi:10.1111/cgf.142662. 3

[SE10] SEGOVIA B., ERNST M.: Memory efficient ray tracing with hierarchical mesh quantization. In *Proceedings of Graphics Interface* (2010). URL: https://dl.acm.org/doi/10.5555/1839214.1839242. 2, 8

[YLM06] YOON S.-E., LAUTERBACH C., MANOCHA D.: R-lods: Fast lod-based ray tracing of massive models. *The Visual Computer: International Journal of Computer Graphics 22* (2006). doi:10.1007/s00371-006-0062-y. 2